An Overview of ‘Upstream at Twilight’

Dublin City University
CA686 Foundations of Artificial Intelligence
Patrick Travers 21267100
patrick.travers3@mail.dcu.ie
17/11/2021

There are two ‘Upstream at Twilight’ programs: Chaser and Trapper, both of which involve a hungry bear and a nervous salmon. Both take place in an identical environment.

Environment definition

A 50x50 square grid (co-ordinates denoted with i and j) including a walled perimeter, where a random 1/3 of the grid is occupied by immovable blocks. There is one agent and one enemy, both of which occupy a single block at any one time. The enemy and agent are capable of movement. The agent may move up, down, left, or right. For the enemy, these movements as well as diagonal movements are permitted. A ‘step’ is defined as any opportunity the agent has the make a move, where it can also choose inaction. The enemy gets an opportunity to make a move only on every second step. The enemy can also choose inaction. The agent and enemy begin on random unoccupied squares.

In this document, capitalised Chaser or Trapper refers to the programs, while ‘the chaser’ or ‘the trapper’ will refer to each program’s enemy.

On the agent’s step, if it is within one square (including diagonals) of the enemy, that counts as a bad step. A step not meeting this condition is defined as a good step. The agent’s score is given by good steps as a percentage of all steps.

A ‘hole’ refers to any configuration of blocks matching the pattern in Figure 1 below. An ‘entrance’ refers to the unoccupied square adjacent to a hole.

If the agent enters a hole and the enemy then moves onto the entrance, the agent is trapped, automatically receiving a score of 0%.

Figure 1. Dark squares represent blocks while empty squares represent the absence of a block. The hole is marked with ‘H’ and entrance with ‘E’. Note that a hole will have only one entrance which may be above, below, or to either side.

Task definitions

Chaser
The chaser's aim is to minimise the agent's score by staying within one square of the agent as much as possible.

Trapper
The trapper aims to trap the enemy in a hole as quickly as possible.

Program assumptions

It is assumed in the general that the enemy and the agent remain ignorant of eachother's algorithms. Trapper assumes the agent has a viable path to at least one hole. If this is not the case, the program will not function. This assumption could be described as a bug as a fix could be implemented to handle this scenario.

Chaser: program overview

The chaser uses an A* algorithm to find the shortest path to the agent. The chaser attempts to stay within one square of the agent as much as possible. Note, throughout this report parenthesised numbers refer to relevant code line numbers.

Classes
Node(440): holds a node's location, it's parent's location, and G, H and F scores (explained further below).
Graph_obj(459): Object to hold nodes. Has methods for checking whether a node is present and for adding a new node.

Search algorithm
Chaser's A* algorithm can be described as follows:

1. Beginning from the enemy's square, identify all adjacent available squares and create a node in Graph-obj for each (578).
2. Calculate each new node's score (F) by summing the node's heuristic distance from the agent (H) and and the number of steps to reach that square from the agent's starting position (G) (517-518).
3. Of all nodes in Graph-obj, identify the minimum F value (601).
4. For all nodes with this F value, identify their adjacent available squares for exploration (604-609).
5. Continue looping from step 2 to step 4 until a node with H value of 1 is found (527).
6. Beginning from this node, traverse a path back to the enemy's position by looking at each successive node's parent (531).
7. Once the path to the enemy is found, identify the enemy's first step along that path (533).

8. Move the enemy to that square (534-535).

Best-first search is used in the fact that nodes with the best F score are given priority in exploring potential paths (601-609). The algorithm meets A search criteria on the basis that a node's score is found by summing G and H (518). Given that the chaser's heuristic (482-484) never overestimates the true number of steps the between enemy and agent, the algorithm meets A* criteria, and is guaranteed to settle on the path requiring the fewest steps.

Heuristic function (482-484)
The chaser's heuristic calculates the larger of the i-distance and the j-distance between enemy and agent. Although Euclidean distance may appear a more informed heuristic, it is actually impermissible. Where the path requiring the fewest steps is sought, a Euclidean heuristic is not guaranteed to find the optimal solution. This is because it unnecessarily penalises diagonal movements. Steps are the effective units of the chaser's actions; Euclidean distance does not account for this. For example, consider that the chaser may move from position [1,1] to [48,1] in 47 steps. Equally, the chaser may move from [1,1] to [48,48] in 47 steps. Euclidean distance of the former path is 47 units, while the latter is 66.47 units, which is at odds with the effective path length for the chaser. Figure 2 below gives a practical example of a scenario where a chaser's heuristic finds a fewer-step path than a Euclidean heuristic.

Important data structures: unexplored_nodes and unexplored_nodes_holder

The objects unexplored_nodes (UN) and unexplored_nodes_holder (UNH) are key to the overall functioning of the algorithm as they facilitate best-first search. When a node is added to Graph-obj, it is also added to UNH (539-542). UN and UNH are identical in structure. Their difference lies in their use-cases; UNH is just used as a buffer object to feed UN (621-634). These objects store node F scores as keys and node locations as values. As the algorithm continues exploring, the locations listed under the least key in UN are given priority (601-609) thus giving best-first search. Furthermore, UN and UNH facilitate a search stopping condition; when both structures become empty, it means the entire available search space has been exhausted without success (615-618). In this scenario, it is appropriate to return a negative result and end the program.

Measures to facilitate computational efficiency

Graph_obj.graph is structured so that nodes are stored as nested objects. The outer key denotes a node's i component. Each i-component object then contains j-component keys. For example, a node at [2,3] would be stored as follows: {2:{3:node}}. The reasoning behind this structure in particular is to save on computational resources in future potential works. For further details on this please see Suggested Improvements section below.

Another aspect of the chaser's algorithm designed to save computational resources is the handling of nodes that have already been visited. When the algorithm is considering a square, which turns to be already stored in Graph_obj.graph (having been already evaluated

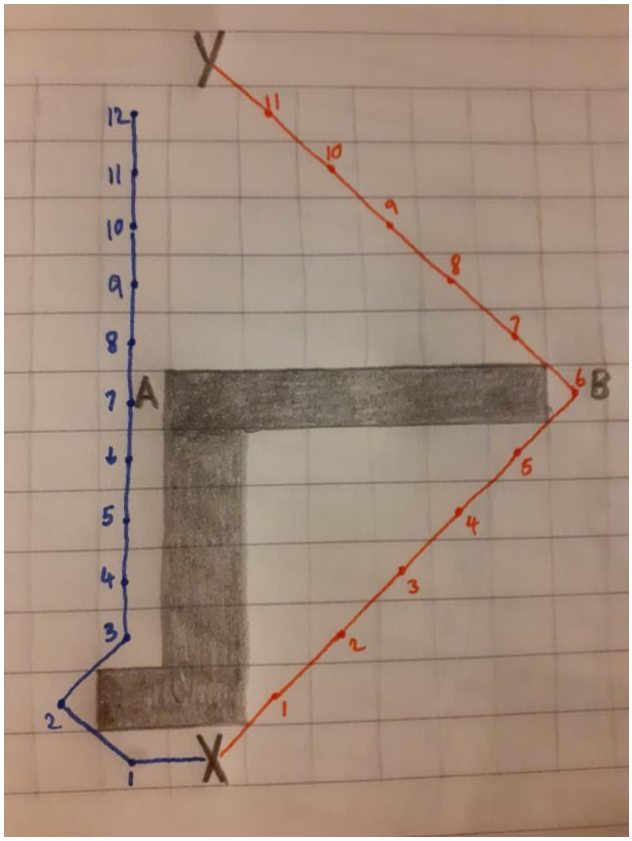

Figure 2. The enemy is stationed at X and wishes to move to within 1 square of Y. A Euclidean distance heuristic would take the blue path taking 12 steps, while the chaser's heuristic would correctly take the red path reaching the target in 11 steps. At position B a Euclidean heuristic would produce a distance of 7.07 to Y, giving an F value of 13.07. At position A, a Euclidean heuristic will give an F of 13. Thus, the Euclidean heuristic will cause the better red route to be ignored in favour of the blue.

along a separate path) that square's existing node will only be overwritten if the current path to that node is shorter than that found previously (556-558). For example, consider the algorithm coming across the square [5,4] at a certain juncture mid-run. Say, this square is already held in Graph_obj.graph. The algorithm will compare the G value [5,4] is due to receive along the current path with the G value of Graph_obj.graph[5][4]. If a faster route has been found, the parent node of [5,4] is updated to the shorter path. Otherwise the algorithm makes no changes to that node.

Algorithm performance

On 104 runs the chaser never scored greater than 0%; the agent was trapped on all 104 occasions. This was achieved in a median of 71 steps, with a standard deviation of 111. Figure 3 below shows a histogram detailing how many steps Chaser took to trap the agent over 104 runs. It was observed during these simulations that the agent very often enters a hole of its own accord with limited coercion by the enemy. The agent's median score up to the point of being trapped was 82.83%. In recording these results runs where the agent began trapped were skipped, treated as void and ignored.

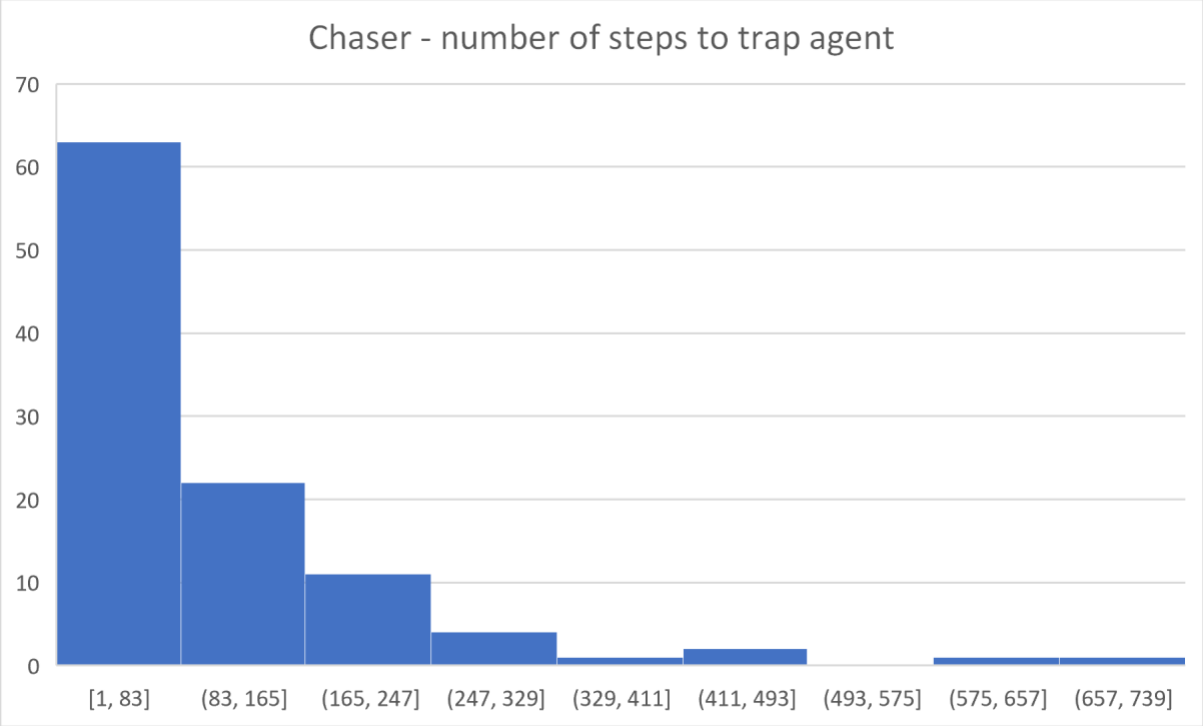

Figure 3. The histogram illustrates the number of occasions on 104 trials that the agent was trapped within the specified step range.

Trapper: program overview

Trapper builds on the framework laid by Chaser and attempts to trap the agent in a hole as quickly as possible. The same A* search algorithm is employed with some adjustments as laid out below.

Search algorithm

The steps involved in Trapper are broadly similar to Chaser, but there are some key differences. Trapper's code can be defined by three broad sections: generic functions (445-516), agent search (527-773) and enemy search (784-995). Generic functions are not particular to enemy or agent search. Agent search functions search find the agent's path to a hole. Enemy search functions find passage to the enemy's optimal square to coerce the agent into that hole. If such a square is unavailable, the enemy reverts to behaviour identical to Chaser. The specific steps involved may be laid out as follows:

1. Starting with squares adjacent to the agent and iteratively expanding the search outwards until success, check squares as to whether they qualify as a hole (722-773).
2. Once a hole is found, check if it is possible for the agent to find passage to it (754-768). This search is an adaptation of Chaser's A*, with a Manhattan distance heuristic to account for the fact that the agent cannot move diagonally (528-530). If no path is available the algorithm will continue looking for another hole, picking up the search from where it left off (as mentioned above, the program works on the assumption there will be at least one viable path to a hole).

3. If the agent has a viable path to the identified hole, the enemy makes it its aim to coerce the agent into that particular hole. This is done using the variable enemy_target_sq. In setting the value of enemy_target_sq, the agent's path to the identified hole is considered. Considering the vector defined by the agent's first step on that path, enemy_target_sq is set to the agent's current location minus that vector (580-588). This is done so that the agent's optimal move in keeping away from the enemy should be to take the next step on the path to the hole.
4. If enemy_target_sq is occupied by a block, Trapper reverts to behaviour identical to Chaser. If the square is free, the enemy's A* search seeks the shortest path onto enemy_target_sq (981-988). On the other hand, if enemy_target_sq is free but the path to it is temporarily blocked because of the agent's position, the enemy will again revert to Chaser's behaviour (928-938).

Interesting aspects of Chaser program outlined above also apply to Trapper. A further point to note in Trapper's code is that different measures are taken throughout depending on whether the enemy is reverting to Chaser's behaviour or trying to reach enemy_target_sq (784-962). For example, there are two separate heuristic functions for each case (785-792).

Failed efforts

The initial implementation of Trapper took a different approach for finding holes for the agent. It attempted to first find all holes in the entire grid and then on each run calculate the agent's heuristic distance to each to find the nearest hole. However, it was decided that a comparable result could be achieved with much better computational efficiency with the present solution. This is due to the fact that at any given time there will generally be a hole within just a few squares of the agent – it is wasteful to continue calculating the agent's distance to distant holes.

Algorithm performance

On 104 runs the chaser never scored greater than 0%; the agent was trapped on all 104 occasions. This was achieved in a median of 84 steps, with a standard deviation of 89. The agent's median score up to the point of being trapped was 74.68%. In recording these results, runs where the agent began trapped were skipped, treated as void and ignored.

Some observances were noted on Trapper's behaviour over these 104 runs. On three occasions the enemy and agent began very near to eachother, with a hole nearby. On these runs, the agent was trapped almost instantaneously (4 steps on each occasion). The same tendency was not observed in Chaser. On three occasions, the real strength of Trapper was exhibited where the agent found its way into what you might describe as a corridor to a hole. In this restricted setting, the trapper effectively marched the agent through the corridor, forcing it into the hole, and executed the trap.

There was one run which uncovered a bug was also ignored in these results. In this run, the enemy deviated from expected behaviour. The enemy stood stationary for hundreds of steps until time ran out while the agent moved back and forth just a few squares away. This seems strange, as the only scenario where Trapper is programmed to remain stationary is when enemy_target_sq has been reached, which was not the case here. As this is due to a coding bug rather than algorithm design it is ignored in evaluating

the performance. Figure 4 below shows a histogram detailing how many steps it took to trap the agent over the 104 runs.

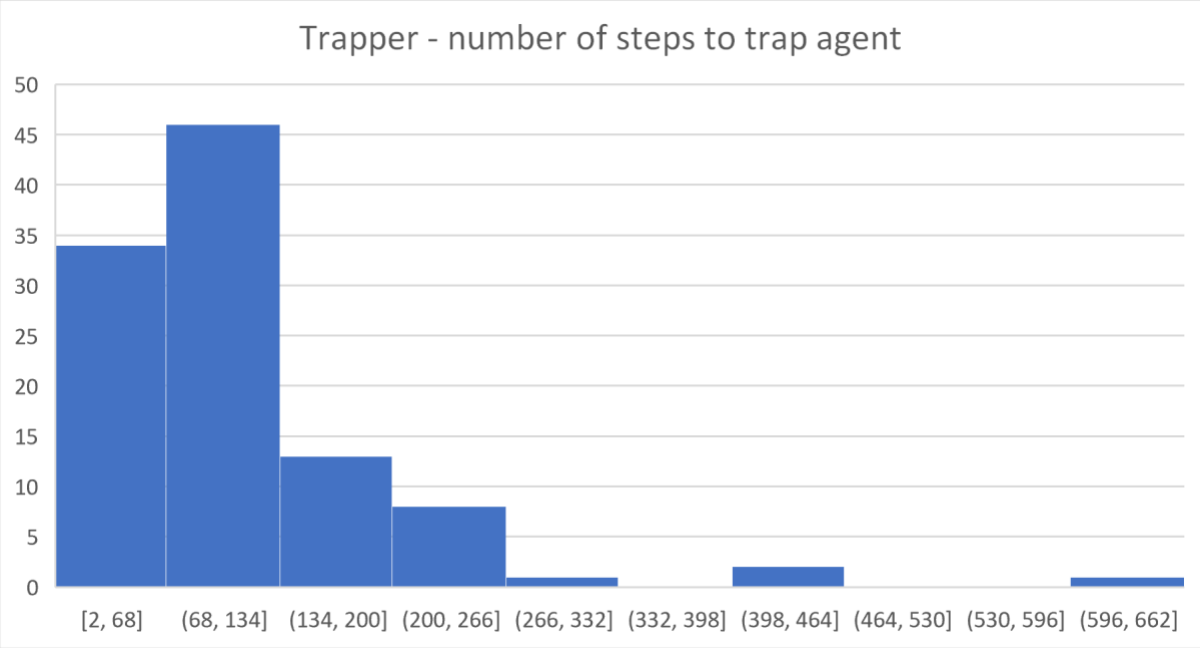

Figure 4. The histogram illustrates the number of occasions the agent was trapped within the specified step range.

Comparing chaser and trapper performance

The first striking different between Chaser and Trapper is that Chaser actually outperformed Trapper in trapping the agent (median step trapped 71 versus 84). However, a Mann-Whitney U Test failed to establish a statistically significant difference between the programs' trapping steps. Up to the time of trapping, Chaser's agent score (82.83%) was slightly worse than Trapper (74.68%). The report now offers a discussion on these results.

The first question to address is why Trapper failed to outperform Chaser on the very parameter it was designed to improve. Some potential reasons are outlined presently.

There is a large element of chance involved in each program. Depending on the starting locations of enemy and agent, it can take enemy in the region of 80 steps to reach the agent's general vicinity and begin its trapping efforts. Furthermore, on occasions where the agent finds itself in an isolated pocket with few blocks, no matter how the enemy might behave, it becomes very difficult to execute a trap. Relatedly, the enemy's coercive power over the agent is limited. The enemy can take a position to encourage the agent to take a path towards an identified hole, but the element of chance involved in the agent's behaviour can lead it to decide to on something completely different. These stochastic factors make it difficult to improve on Chaser's behaviour, where most of the time the agent already traps itself of its own accord.

As mentioned in Trapper's algorithm performance above, there were only 3 occasions out of 104 runs where, on casual observance, Trapper had opportunity to properly exhibit its coercive strength. This is too small a proportion of runs for any algorithmic improvements to have an influence on overall results.

Although this report considers that Chaser and Trapper perform really about the same (discounting Trappers noted bug), there is one tendency of Trapper's which might be described as a disimprovement on Chaser. On casual observance, it seemed Trapper was sometimes prone to getting stuck in loops while still at a distance from the agent, where it appeared to be changing its mind back on forth as to which path it should take. This is a natural consequence of Trapper's conditional targeting of either enemy_target_sq or the agent itself. Since trapper's coercive power is ultimately limited, this added inefficiency is perhaps not worth the additional effort of concertedly aiming towards enemy_target_sq.

Any advantage Trapper has over Chaser in agent score at time of trapping is likely explained by the fact that Trapper took slightly longer on average to trap the agent. The length of run in steps and agent score for both Trapper and Chaser exhibits strong negative correlation.

Suggested improvements

As outlined above, there is a bug which causes Trapper to inappropriately remain stationary on occasion. This behaviour is unintentional and at present the cause is not known. Furthermore, Trapper could be amended to produce an appropriate error message in the case where there is no hole available to the agent. The function draw_path is incomplete and non-functional which should be amended.

Although Trapper succeeds in identifying a hole where one is available, it makes no appraisal of how difficult it is going to be to coerce the agent into that hole. A hole must have an entrance, but if the square one step further beyond the entrance is occupied, it becomes more difficult to coerce the agent into that hole. Trapper could account for this when choosing a hole for coercion. In the current implementation, computational resources are wasted in finding a hole in that previously checked squares are check again and again unnecessarily (741-751). Relatedly, Trapper could choose to favour zones with many available holes rather than aiming for one hole in particular. Further still, squares which form entrances to two or three holes could be favoured over entrances which lead to only one hole. The definition of a hole could be expanded to incorporate ideas such as a 'corridor', as Trapper was observed to perform particularly well within such structures.

Once a hole has been found, Trapper decides on the particular path it wishes the agent to take. However, there may be many viable paths towards that hole. Once a hole is identified, Trapper should instead be flexible on the specific route for coercion rather than insisting on any one route in particular, even ignoring objectively optimal routes where appropriate. A separate adjustment with potential to improve Trapper would be to search for another hole when enemy_target_sq is occupied, rather than reverting to Chaser behaviour. Rather than searching for the closest hole by a heuristic measure, the shortest real path from agent to hole could be calculated.

Another relatively simple change could be made to Trapper to save computational resources – there is little gained in aiming towards enemy_target_sq when the enemy and agent are on opposite sides of the grid. The program could be adjusted so that enemy_target_sq only becomes an aim when the enemy comes within touching distance of the agent. Another option when the enemy must traverse the entire grid to reach the agent is to try to anticipate where the agent is likely to be by the time the enemy is arriving which may be up to 80 steps in the future. Incorporating this stochastic element may aid the

enemy in reaching the agent faster. The present path-finding algorithm behaves as if the agent will remain in situ while the enemy completes its path in full.

Computational resources could be saved by initially calculating whether the agent's position is principally in the i or j direction. Because Graph_obj.graph is constructed with nested objects, checking if a node is already in this structure would be optimised by ensuring that the outer nest signifies the direction in which the most ground needs to be covered in seeking the agent. At present, Graph_obj.graph stores nodes in an [i][j] structure, even if the agent lies principally in the j direction. Taking account of the agent's principle direction should offer some computational savings.

Potential future works

Given the coercive power of one enemy is quite limited against an agent, an interesting project may be to attempt the same project with two or three enemies. Three enemies should be capable of constructing a 'hole formation' to encapsulate the agent and apply a strong coercive force towards a hole.

Given the above observation that a Euclidean heuristic is impermissible in this particular problem, it would be interesting to build a similar grid, replacing squares with regular octagons. In this scenario, a Euclidean heuristic would be permissible as step length would be direction-invariant.